SIXTH FRAMEWORK PROGRAMME

PRIORITY 2 IST-2005-2.5.5

Software and services

SHADOWS

Deliverable D3.2.2: Prototype for self-healing concurrent development

Project acronym:	**SHADOWS**
Project full title:	**A Self-healing Approach to Designing Complex Software Systems**
Contract no.:	035157

SHADOWS is a Project funded by the EU under contract No. 035157. The Shadows consortium includes IBM Israel - Science and Technology Ltd, Universit´a degli studi di Milano - Bicocca, Universit¨at Potsdam, Vysoke Uceni Technicke v Brne, Royal Philips Electronics, Comverse Ltd, ARTISYS, Net Technologies Ltd, Scapa Technologies Ltd.

	SHADOWS – REPORT ON WORK PERFORMED AS PART OF TASK T3.2 **Deliverable. no: D3.2.2**	
IST-2006-35157	**Version: 1.0**	**28/2/2007**

Deliverable type: Report

Classification: Pub.

Work package and task: **WP3, T3.2, D3.2.2**

Responsibility: **HRL**

Executive summary:

This document describes the prototype tool for self-healing of concurrency-related failures for the development stage. The prototype utilizes the APIs resulting from Task 3.1 to provide automated testing, failure analysis through static and dynamic techniques, and preliminary healing capabilities. It will be iteratively improved in the following deliverables.

The report describes project activities undertaken as part of task T3.2

Table of Contents

TABLE OF CONTENTS 3

1. INTRODUCTION 4

2. THE PROTOTYPE TOOL COMPONENTS 4

2.1. The Testing Component 4

2.2. The Failure Analysis Component 6

2.3. The Healing Component 8

3. USER MANUAL 8

3.1. Instrumentation 8

3.1.1. Checking instrumentation status 9

3.1.2. Method size limit 10

3.2. Running Tests with ConTest 10

3.2.1. Output 10

3.2.2. Verbose output 11

3.3. Heuristics 11

3.3.1. General noise options 11

3.3.2. Focusing noise on shared variables 12

3.3.3. Halt-one-thread heuristic 12

3.3.4. Tampering with timeout 13

3.3.5. Deferring noise generation to a late point in the execution 13

3.4. Listener API Description 14

3.4.1. Which events can be listened to? 14

3.4.2. Which events currently *cannot* be listened to? 15

3.4.3. Multiple events 15

3.4.4. How to register listeners? 15

3.4.5. Deferred registration 16

3.4.6. Things you should know when implementing listeners 16

3.5. Infrastructure for Search Algorithms 18

4. SUMMARY 19

1. Introduction

Workpackage 3 is responsible for defining and developing the SHADOWS technology for healing concurrency problems. In order to heal concurrent bugs, it is first necessary to identify the root cause of the bug. In a previous deliverable (3.1.1), we described the interfaces for test drivers that schedule the execution of program threads to generate interesting scenarios, and test oracles that provide indication whether a program run failed.

The purpose of this document (Deliverable 3.2.2) is to describe how these interfaces were utilized to develop a prototype tool for self-healing of concurrency-related failures for the development stage. The underlying algorithms are described in Deliverable 3.2.1.

Deliverable 3.2.2 is part of Task 3.2 – development of self-healing tool for the development stage. The task will generate deliverable D3.2.1-5, which will iteratively improve the prototype using static and dynamic techniques for locating failures and for checking the safety of the program modifications.

2. The Prototype Tool Components

A tool for self-healing of concurrency-related failures should contain the following components: testing, failure analysis, healing and verification of healing. The prototype tool developed thus far contains testing, failure analysis and preliminary healing capabilities.

2.1. The Testing Component

The prototype is built on top of the testing component ConTest, an advanced testing solution from IBM, whose main use is to expose and eliminate concurrency-related bugs in parallel and distributed Java programs. ConTest is developed and supported by IBM's Verification and Testing Solutions group in Haifa. ConTest systematically and transparently schedules the execution of program threads such that program scenarios which are likely to contain race conditions, deadlocks and other intermittent bugs - collectively called synchronization problems - are forced to appear with high frequency. To enable easy use of ConTest, we have created a ConTest plug-in for eclipse 3.1. A user can now instrument classes and configure ConTest properties from within the eclipse IDE.

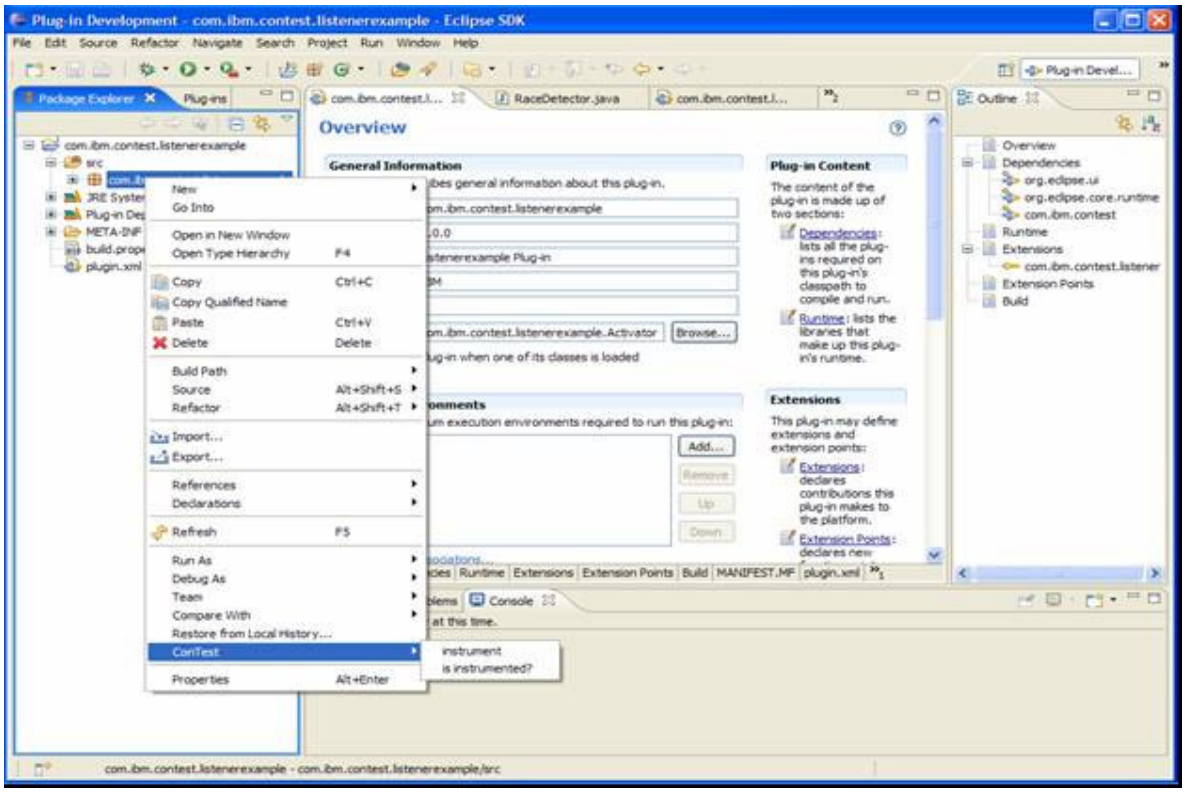


Figure 1: Instrumentation using ConTest Plug-In

Figure 1 presents the ConTest plug-in options to instument classes and check whether the project classes are instrumented, by choosing the corresponding options from the right-click menu.

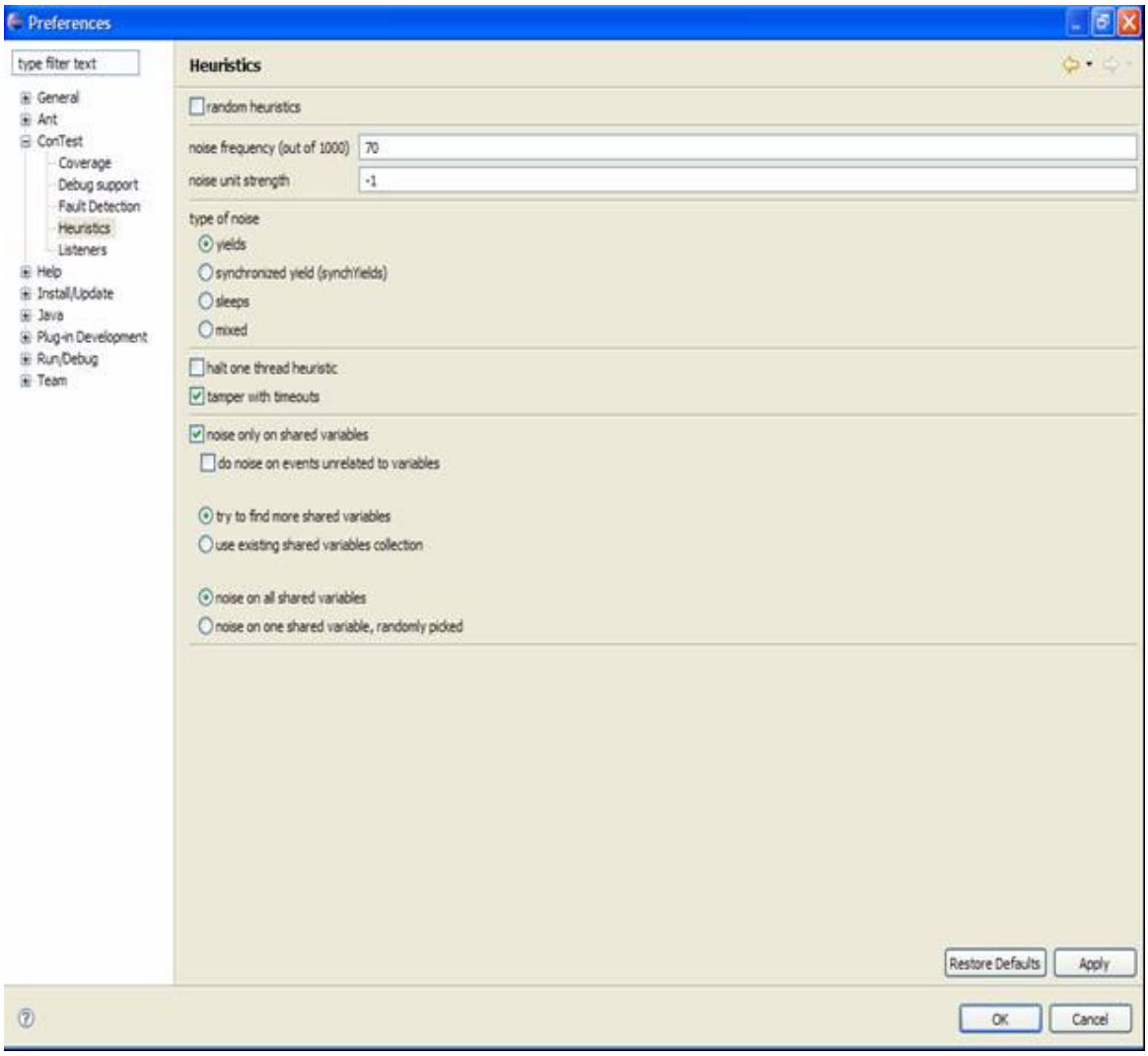


Figure 2: Configuring ConTest properties through ConTest Plug-In

Figure 2 presents ConTest's configuration window, through which the user can control the settings of the tool.

2.2. The Failure Analysis Component

For the failure analysis phase, a race detector was implemented by the partners from Brno University, using ConTest Listener architecture which is described in Deliverable 3.1.1. The race detectror implements a modified version of the Eraser algorithm for race detection, and is described in details in Deliverable 3.2.1. It was run on small Java Programs and was able to detect true races. The code of the race detector was added as an Eclipse plug-in extension to the ConTest plug-in.

Figure 3: Adding a Listener Plug-In to ConTest Plug-In

Figure 3 presents how the extensions tab of the Eclipse plug-in xml editor is used to define a new Race Detector listener as an extension to the com.ibm.contest.listener extension point.

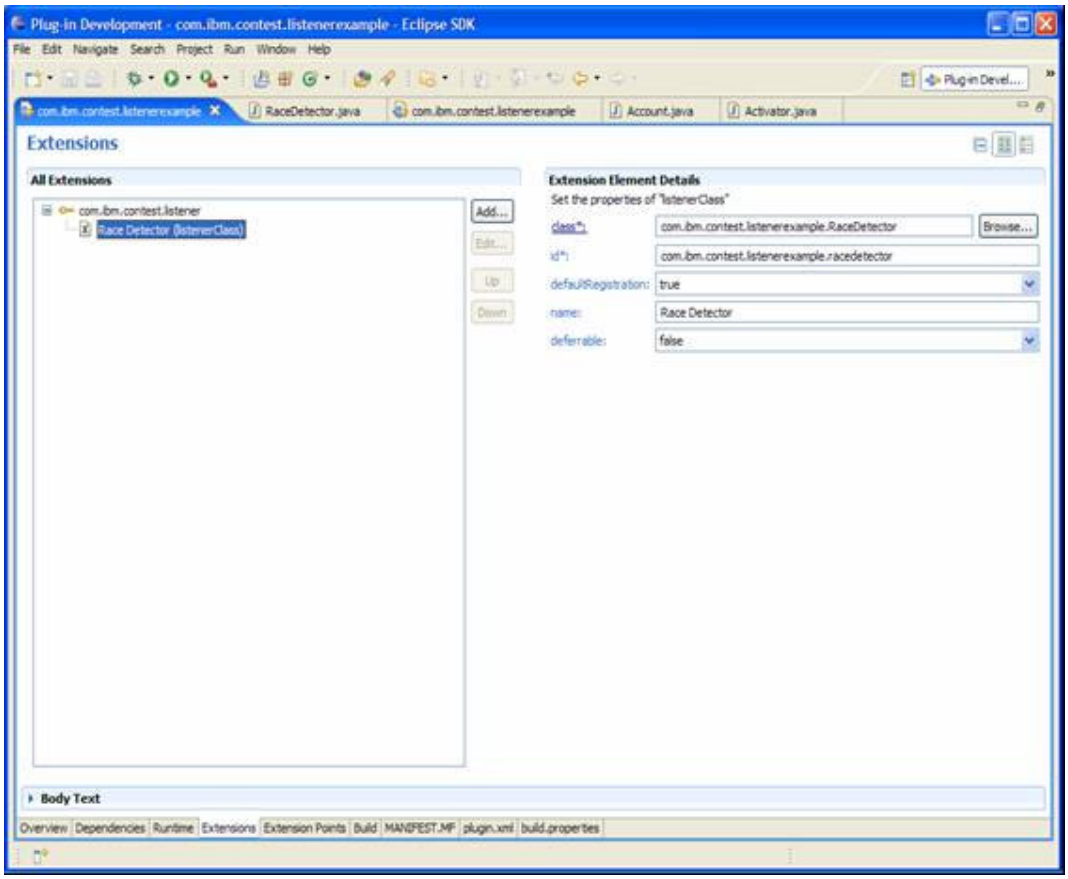
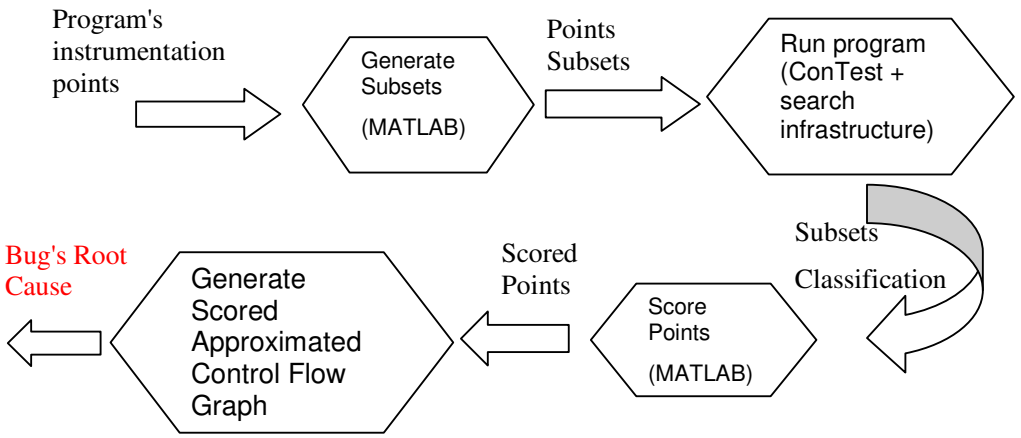


Figure 3: Adding a Listener Plug-In to ConTest Plug-In

A different approach for failure analysis was used when implementing an automatic debugging technique using ConTest's infrastructure for search algorithms for minimal noise. The technique uses the infrastructure in order to run the program under test with many different subsets of instrumentations (locations in the code in which noise is added), classify each subset as either inducing or not inducing the bug, and use feature selection techniques from the domain of machine learning in order to score the program locations, and correlate the locations with the highest scores to the root cause of the bug. The flow of the technique is as follows:



The automatic debugging technique was tested on two real-life Java programs, and in both cases was able to pinpoint the root cause of the bug.

2.3. The Healing Component

The work on healing is still in a preliminary stage. Healing was implemented on top of the race detector, using two different approaches:

1. Affecting the scheduler to avoid an undesired context switch - this approach decreases the probability of a context switch within a specific segment of code to maintain its atomicity. It can be done either by temporarily changing threads priorities, or by calling yield() just before the segment code in question.
2. Adding an explicit lock to avoid a detected race condition.

The healing algorithm is described in details in Deliverable 3.2.1. The two healing approaches were tested on a small Java program that contains a race condition with partial success.

3. User Manual

ConTest is a powerful, extremely easy to use tool for finding bugs caused by concurrency earlier in the testing process. It alleviates the need to create a complex testing environment with many processors and applications, and works by instrumenting the bytecode of the application with heuristically controlled conditional sleep and yield instructions. ConTest is useful to, and used by, both testers and developers. ConTest is available as either a stand-alone application, or a plug-in for the eclipse environment.

ConTest can be rapidly employed with minimal effort, and used to measure coverage, to aid in debugging, and to present deadlock information. ConTest has a variety of additional features, most of which were implemented due to user requests. To use ConTest you need to instrument your application and rerun your multi-thread or distributed tests many times.

3.1. Instrumentation

When you test with ConTest, the first step is instrumentation. The application class files are modified so that its execution will be more likely to exhibit bugs. Additional modification can be made for coverage or other needs.

To instrument your class files located under a root directory **root**, go to **root** at the command line (cd to **root**) and enter, in one line:

```
Java -classpath
"c:\JavaConTest\Lib\cfparse.jar;c:\JavaConTest\Lib\ConTest.jar"
com.ibm.contest.instrumentation.Instrument .
```

or for UNIX:

```
java -classpath
"/home/JavaConTest/Lib/cfparse.jar:/home/JavaConTest/Lib/ConTest.jar"
com.ibm.contest.instrumentation.Instrument .
```

(These commands assume that you unzipped ConTest in c:\ or in /home/. Also, note that the commands include a '.' at the end to indicate the current directory.)

ConTest recursively instruments the entire tree of classes and backups the original classes. If a class is named className.class, the original class is saved under className.class_backup and className.class becomes the instrumented class.

You can use a similar command to instrument an entire jar file. Instead of specifying a directory in the command above, specify the name of the jar file that you wish to instrument. A backup copy of the jar file will be created and all the classes in the jar file will be instrumented.

To instrument a specific class file or files, enter the file name or list of files at the command line. The files in the list can be separated by colons (UNIX), semicolons (Windows) or spaces (all systems):

```
java -classpath
"c:\JavaConTest\Lib\cfparse.jar;c:\JavaConTest\Lib\ConTest.jar"
com.ibm.contest.instrumentation.Instrument Class1.class Class2.class
..\anotherDirectory\Class3.class
```

(for windows; compare the example above for UNIX)

In addition, you can use the following flags in the instrumentation process:

`-nobackup` - backup files are not saved.
`-norec` - when a folder is specified, folders are not instrumented recursively.
`-exclude (file)` - do not instrument classes listed in the file (one class per line, fully-qualified class name).
`-only (string)` - instrument only classes whose fully-qualified name contains the string.
`-verbose` - verbose instrumentation output.

There are other flags, used for coverage.

3.1.1. Checking instrumentation status

To check whether a class (or a directory, or a jar) is instrumented, invoke the instrumentation command with a flag -isinst. If you invoke it for a directory or a jar, it reports how many classes are instrumented and how many are not (and, rarely, how many are partly instrumented - see below). A directory is checked recursively, unless -norec is also specified. No other instrumentation flag is applicable with -isinst.

3.1.2.Method size limit

The JVM poses limitation over the size of methods (65536 bytes in class file representation). It might happen that a very long method is originally legal, but after instrumentation it would exceed this limit. Such methods are not instrumented, and a warning is printed. If instrumentation status is checked for a class containing such method(s), it will be reported as "partly instrumented".

If you want to instrument such a method, rewrite or split it so that it is smaller.

3.2. Running Tests with ConTest

Testing does not have to be modified to accommodate ConTest. The instrumented application is tested in exactly the same way.

To use ConTest, JavaConTest/Lib/ConTest.jar must be located in your classpath.

The **KingProperties** file (also taken from directory Lib) should be available, in **one of four ways** (from the higher to the lower priority):

1. Specified using -DconTestPropFile=<file path> command line argument (JVM property) when running the instrumented application.
2. Located in current working directory, where you run the instrumented application
3. Specified in your classpath
4. Located in the same folder as jar files you've added to the classpath

Note about specifying path in a JVM argument: In some Windows applications, if the path contains backslashes, you have to double them so that they are not taken as escape sequences: -DconTestPropFile=d:\\some_dir\\contestProps.txt. If you are not sure, just try and see what works. The verbose output (see below) would make it very clear.

You are now ready to run your tests.

3.2.1. Output

All ConTest output files (in particular, coverage files) are written to a directory com_ibm_contest (we will refer to it as "ConTest output directory"). Some files are written to subdirectories of that directory. This directory itself is located by default at the current working directory. If you want it inside another directory, change the output property in KingProperties, and use -outputDir instrumentation option, to specify the desired location for the output directory. Sometimes the tests are not run from the same directory as the instrumentation; using default output directory would then result in two different directories. This state is not forbidden, but is less convenient for viewing coverage information. In such a case it is recommended to change the output directory either in instrumentation, or in test runtime (KingProperties), or both, to point to the same directory.

Note that in Windows, a backslash between directories must be doubled when given in the property. For example, output = c:\\foo\\bar
For ConTest to work properly you need to have writing permissions in the directory you choose (or in the current working directory, if you didn't specify otherwise).

Run ID: Output files which relate to a given run (coverage trace files, replay files) are named with an ID of the run. By default this ID is a timestamp (a long number - millis since epoch). You can set your own ID, so that it will be easier for you to identify files from a given run. This is done by adding to the command of the test run a JVM argument "-Dcontest.run.id=xxx", where xxx is your chosen string.

3.2.2. Verbose output

If you set -Dverbose=true on the command line when launching an instrumented application, ConTest will verbosely print some of its major steps (initialization parameters read, output files written, etc).

3.3. Heuristics

A heuristic is the algorithm used to create "noise" and increase the chances of revealing a synchronization bug. ConTest affords a combination of different heuristics, that you can control according to your test needs.

To simplify the usage of ConTest, by default the heuristics options are chosen at random at each test, so if you don't specify otherwise, each test will use a different combination of heuristics. To specify heuristics manually, as described in the rest of this page, you need to turn off the random feature. This is done in KingProperties, by setting the random property to false (don't delete it - if the property is missing, it is considered true!)

Using verbose output allows you to see all choices made by the random feature.

3.3.1. General noise options

ConTest currently supports three different types of noise, specified through the noiseType property in KingProperties: yields, sleeps, and synchYields). They differ by the type of synchronization primitive they use to create the noise (java.lang.Thread.yield() or java.lang.Thread.sleep()). In addition, you can specify mixed in this property, which chooses a different one at each concurrent event.

The amount of noise can be controlled by two properties: noiseFrequency and strength. The first one is a number between 0 and 1000. It determines the probability that at each concurrent event noise will be done. The unit is "out of 1000" - that is, value of 50 means to do noise approximately once in 20 concurrent events, 100 - once in 10, and 1000 - always. The second determines how much noise is done, if noise is done at all, at a given concurrent event. Setting strength = -1 lets ConTest choose a default unit strength according to the noise type. strength = 0 means almost no noise at all.

Usually, by increasing the frequency and the strength, you increase the chance to find bugs. On the other hand, you also increase the runtime cost of the noise. You can play with the noise type, frequency and strength properties, to determine the maximum noise that you can introduce into your application while retaining reasonable performance.

If random is applied (see above), the strength and frequency are chosen randomly, but around the values specified in KingProperties - so you can use random heuristics while maintaining control over the amount of noise.

When using the sleeps and mixed noise type, all invocations of java.lang.Thread.interrupt() and java.lang.ThreadGroup.interrupt() in the application need to be instrumented by ConTest. That is, if you use partial instrumentation, make sure uninstrumented parts of the code don't include interrupt(). If you got a message "HeuristicsSleep: took an interrupted exception:" then probably this rule has been violated.

3.3.2. Focusing noise on shared variables

ConTest can attempt to identify which variables are accessed by more than one thread, and do the heuristic noise only on accesses to those variables.
To activate it, set the property sharedVarNoise = true in KingProperties. A variable is determined to be shared when two different threads accessed it (read or write). In the first run, in addition to identifying shared variables and making noise on them, the variables will be written to a file, sharedVars.txt, in ConTest output directory. In subsequent runs, those variables will be known in advance to be shared, and noise will be made on them from the *first* access.

- With this option on, the heuristic noise can be any one of those described above (sleep, yield, etc.), with any strength. Note, however, that the noise is only done here on a small subset of the concurrent events. This allows you to set much higher strength without severely affecting runtime performance.
- Normally, when the shared variable option is activated, ConTest will attempt to identify more shared variables and add them to the file sharedVars.txt. However, if you set the property collectShared = false, only the variables already in the file in the beginning of the run will be considered as shared.
- If you set the property shared = one, and the sharedVars.txt file exists in the beginning of the run, ConTest randomly selects one variable from the file, and noise is focused only on accesses to that variable.
- Some concurrent events are not variable access - synchronization primitives, methods activation, etc. By default, when shared variable noise is on, there will be no noise on these events. Set the property nonVarNoise = true to cause noise on such events, too.
- If you suspect about a certain variable (or several variables) that it is a source of a bug, you can cause ConTest to focus solely on that variable. Do this by editing the file sharedVars.txt manually, letting it contain only the variable(s) you want, and setting collectShared = false. The file contains one line for each variable. The line contains the full class name, then a period, then the name of the member as it appears in the code. For example, com.ibm.some_project.SomeClass.someMember.

3.3.3. Halt-one-thread heuristic

This heuristic occasionally causes one thread to stop executing for a long time, until no other thread can advance. It can be powerful in revealing some sorts of concurrent bugs. To activate it set haltOneThread = true in KingProperties (and make sure the frequency property is set to more than 0).

Turning busy-waits into hang

In multi-threaded programs, often a thread needs to stop activity until some condition obtains. For example, it could be a thread that does some short service whenever another thread requests it, and rests between requests. An appropriate way to do it is to use Object.wait in conjunction with Object.notify. An inappropriate way is to do "busy-wait":

```
while ( ! conditionIsMet) {
  continue;
}
```

Where conditionIsMet is set by other thread(s); it may be a more complex condition. Such a code is bad, because CPU will be wasted on this empty computation (by contrast, wait relinquishes CPU). In addition, busy-wait loops will not work well if different priorities are used for threads.

ConTest can help detect such busy-waits: if the halt-one-thread heuristic is on (either directly by the property, or indirectly though the random mode), the loop has a high chance of running infinitely. Thus the program will appear to hang. If your program hangs, and you want to make sure that the reason is busy-wait detected by halt-one-thread, press Ctrl-Break to produce Java core dump. Look in the dump file for the "Thread info" section, which gives you the call stack for all the threads. One thread may present a stack such as:

```
3XMTHREADINFO      "Thread-2" (TID:0x92B6B0, sys_thread_t:0x11FF6920, state:R,
native ID:0xB20) prio=5
4XESTACKTRACE          at java.lang.Thread.yield(Native Method)
4XESTACKTRACE          at com.ibm.contest.Noise.makeNoiseYield(Noise.java(Compiled
Code))
4XESTACKTRACE          at com.ibm.contest.Noise.makeLongNoise(Noise.java(Compiled
Code))
4XESTACKTRACE          at
com.ibm.contest.Noise.makeNoiseHaltOneThread(Noise.java:111)
4XESTACKTRACE          at com.ibm.contest.King.beforeConcurrentEvent(King.java:439)
4XESTACKTRACE          at MyClass.myMethod(MyClass.java:10)
4XESTACKTRACE          at MyThread.run(MyThread.java:11)
```

The marked line here is what you should look for. Looking at the stack of other threads, ignoring calls to com.ibm.contest..., and inspecting the source locations, you may see them in busy-wait code.

If you know of a busy-wait in the program and you accept it and don't want ConTest to halt the program because of that, disable halt-one-thread, by setting random and haltOneThread, to false.

3.3.4. Tampering with timeout

The method java.lang.Thread.sleep(long millis) (also exists with additional nanoseconds parameter) causes the current thread to stop executing for the specified duration. While this is useful when you (the programmer) want your thread to relinquish CPU for some time, this time duration alone should not be counted upon: you should not assume that other threads have completed some tasks by the time the sleep returns. It is unsafe, because when the system is heavily loaded, it may take longer for the other threads then you assume. So even if it works in unit test, it may fail in the field. If you need to guarantee that some condition obtains before continuing execution, check this explicitly after the sleep, and if necessary sleep again (in a loop). Alternatively, use another mechanism, like Object.wait() and notify(). Similarly, you should not count blindly on the time-out parameter of Object.wait(long timeout) and Thread.join(long millis).

ConTest helps testing that such wrong assumptions are not done, by randomly reducing the time-out used by these methods. This simulates a condition in which other threads work slower, and thus the bug may be revealed. If you do the right thing - sleep repeatedly and testing a condition - this tampering will not affect the program.

To cancel this behavior of ConTest set both random and timeoutTampering to false in KingProperties. We recommend that you do it only if you have good reason.

3.3.5. Deferring noise generation to a late point in the execution

You might not want ConTest to begin its perturbations just as the tested program begins. For example, maybe you found a bug, using ConTest, in the initialization of your program, but you want to go on testing the rest of the program, without ConTest causing the bug to appear

again and again.
If you know that a certain piece of code - either a certain class, or a certain method, is used only after the part you want undisturbed, you can use this. Set the property beginAtValue to a string which is the class name or the method name. Set the property beginAtType to either class or method, accordingly. If you set one of these two properties, you must also set the other. This sets a condition on the code location of each concurrent event. Until this condition is met, ConTest does no noise. Once it has been met, noise is done according to other conditions (heuristics, shared variables, etc.)

The exact specification for a program location to match the condition is that the class name or the method name in the program location will *contain* the string given in beginAtValue. This means you can either give the full class name or just the private name of the class, whichever is more convenient for you. Note that if you give a certain method name, it will match a method that contains that name in *any* class.

This mechanism also defers coverage collection, debug info collection, and all tracing.

3.4. Listener API Description

This API provides the listeners architecture of ConTest, and the necessary infrastructure. A demo of how to program with listeners can be found in JavaConTest/ReadmeFiles/listenerDemo. The package contains:

- The listener interfaces, which define the points where you can plug-in. Their names all end with "Listener".
- Class Registrar, with which you can programmatically register and remove listeners.
- The rest of the classes are utility classes that provide an interface with various ConTest runtime functionalities. They are useful when writing listeners and you should have an idea about them.
- Note also the package com.ibm.contest.utils, which provide more utility classes. These classes are not specifically related to ConTest runtime, but they are also useful when writing listeners.

The two packages are compatible with Java 1.4 and above.

3.4.1. Which events can be listened to?

- Thread begin and end
- Operations relating to synchronization: obtaining and releasing monitor, wait and notify
- Other operations relating to thread lifecycle: start, join and interrupt
- Reads of, and writes to, variables. These include simple variables and array cells. For simple variables, only member variables (instance and static), but not method local variables, can be listened to. For array cells, there is no such distinction. This is explained in more detail in BeforeArrayCellWriteListener.
- Entry to methods
- Entry to basic blocks
- Test reset (a user's decision, given explicitly to ConTest)

Typically one class will implement several listeners to achieve a certain task. You just need to register its instance(s) several times, one for each type of listener. It is possible to register the same listener object more than once as the same listener type. In this case, the code will be

invoked more than once for the same event. However, we don't know of a good reason to do so.

3.4.2. Which events currently *cannot* be listened to?

We don't define listeners for the methods suspend, resume and stop of Thread, which relate to thread lifecycle but are deprecated (and indeed hardly used). Let us know if you do want listeners for them.

Some events relating to threads interrupt status can be listened to: interruption and taking of InterruptedException by wait and join. However, not all: taking of the exception by sleep, and turning off interrupted status by interrupted () are currently not listenable. Let us know if you want it.

Calls to java.util.concurrent (from Java 5) cannot be listened to, although many of them fall into categories of the defined listeners. We will add support for it in future versions.

There is no listener to the end of the program. You should not assume that the JVM running the test will terminate normally - many testing setups kill the JVM abruptly. If you want some code to be executed when the JVM terminates, use shutdown hooks (see java.lang.Runtime.addShutdownHook). Note that shutdown hooks have their limitations, given that typically you don't know how the target program will terminate; it's usually better if you can do without them. For example, if you use output files and flush them often enough, you'll be safe even if the JVM terminates without their orderly closing. In any case, ConTest will probably not be able to supply something superior to JVM shutdown hooks.

3.4.3. Multiple events

Usually one runtime event is related to at most one type of listener. Notable exception is method beginning: in addition to MethodEntryListener, it triggers BasicBlockEntryListener. In case this method is a thread's main method, it triggers also ThreadBeginListener. In case it is synchronized, it triggers also BeforeMonitorEnterListener. The last two conditions are usually mutually exclusive, but not necessarily: Thread.run() and main() *can* be declared synchronized. The order is:

1. ThreadBeginListener
2. BeforeMonitorEnterListener
3. *monitor-enter*
4. AfterMonitorEnterListener
5. MethodEntryListener
6. BasicBlockEntryListener

If a thread's main method is synchronized, BeforeMonitorExitListeners, *monitor-exit* and AfterMonitorExitListeners will be executed before ThreadEndListeners.

The order of invocation among multiple listeners registered for the same event type (instances of the same listener interface) is not specified. Let us know if you need a stronger guarantee.

3.4.4. How to register listeners?

Your class which implements the listeners should normally have a no-argument constructor. ConTest will create an instance of this class and will start passing events to it. ConTest is told about listeners in XML files. These XML files must be in a listeners' directory which must be

in the same directory as KingProperties file which the user uses (see ConTest readme about properties file). Note that the properties file, and hence its directory, may be in Lib under ConTest installation directory, or in classpath, or pointed to by a JVM argument. The Lib directory contains an example registry file extensions.xml.example, which you can take as a template for your own XML. The only mandatory field in this XML is your class's fully-qualified name.

You can register several listener classes in the same XML. You then give this XML to your users together with your classes, and instruct them to add the XML to the listeners directory located next to the KingProperties they use. Note that typically this directory may contain other XMLs for listeners written by other providers.

In addition to registering listeners by XML, you can also register them programmatically by the Registrar class. The methods of this class get an instance of the listener class. Normally you should use this class only when ConTest is already running - otherwise it will cause ConTest to initialize itself, and that may have several side-effects. That is the main reason to use Registrar - to add listeners at a late point in the program-under-test's (and hence ConTest's) execution (you can't do that with listeners registered by an XML). Another scenario in which you can't use the XML registration and must use Registrar is if you can't have a no-argument constructor. The Registrar also allows you to un-register listeners.

3.4.5.Deferred registration

The user can tell ConTest, through properties, to start activity at some late point in the test - currently the options are when a certain method or class are run. Usually your listener should respect that, unless it's crucial for the listener's correctness that it's active from the start. You don't need to worry about that in your event handling code; rather, in the registry XML, or in Registrar, you define whether or not your listener is deferrable. If it is, it will start getting events only when the condition for starting activity is met (and will cease getting events if the test is reset, and so on repeatedly). Registrar allows you the further flexibility to register your listener class as deferrable for some even types and non-deferrable for other types.

3.4.6.Things you should know when implementing listeners

- Many listeners get a programLocation string parameter. This string contains several fields describing the point in the code which caused the event. Class com.ibm.contest.utils.ProgramLocationParser enables you to extract these fields. To spare redundancy, the programLocation parameter is not documented in the individual listener documentation.
- Program locations are unique: ConTest guarantees that if p1 and p2 are two program locations, then p1.equals(p2) iff p1 == p2 (provided, of course, they were handed directly to the listener by ConTest, and not created by means of cloning, serialization, etc.). You can use this fact to improve performance.
- Strings representing class names and field names, given to field access listeners, are unique if and only if they come from the code of one given class: Suppose two field accesses see class names *className1* and *className2*, and *className1*.equals(*className2*). If the two accesses are in the code of the same class, then *className1* == *className2*. If the calls are in different classes (possible if at least one of the accessed fields is not private), the references will be different, although they truly represent the same class.
- There are many listeners which form pairs of *before* and *after* a given type of event.
    - The documentation of the *after* always refer to that of the *before*.
    - The *after* method gets all the context-dependent arguments which the *before* gets. They are documented only in the *before*.

    - Usually the thread will perform the sequence *before*-event-*after*, but not necessarily:
        - If any *before* listener throws an unchecked exception, then the event and the *after* listeners will not be executed (see below).
        - If the event itself (the bytecode operation) throws an exception or an error, then the *after* listeners may or may not be invoked, according to the following rules:
            - In variable and array access events, and in monitor-enter and exit, all throws cause skipping of the listeners. The conditions under which this can happen are documented in each *after* listener, as "Conditions for missing invocation".
            - In listenable java methods, exceptions will not cause skipping. The listeners get the exception as argument, for information. After the listeners run, the exception is thrown.
            - Errors thrown by all events will cause skipping. Usually you don't need to give consideration to this possibility, since the target program is not expected to continue running after an Error.
- Unless you know something special about the instrumented application, all listener code should be expected to run in parallel.
    - In particular, a thread switch can happen at any point between the *before*, the event and the *after*, and other listener code can run (or code of the same listener by other threads). With some events (e.g. monitor-enter) this is even very likely. ConTest adds no synchronization when invoking the listeners: use synchronization inside your listener code as necessary. However, registering and removing listeners *is* synchronized by the manager.
    - If you do all your listener registration in contestInitEvent(), you don't need to worry about parallel effects between registration and invocation: the registration is guaranteed to complete before any listenable event occurs.
    - If, conversely, you register or remove listeners in a later occasion, there is a delicate point here: an event to which this listener listens may happen in another thread in parallel. In this case, the listener may or may not get this event. In architectures which truly implement two-layer memory model (some multicore systems), it can even happen that an event occurring after the registration (removal) will be missed (taken) by the listener; such anomaly is possible as long as the thread in which the event happened has not done synchronization. For more specific details on what is guaranteed here, consult the memory model discussion in the Java language specification. The runtime code still guarantees good behavior: the event may or may not be seen by the listener, but otherwise there would be no bad effect to parallel registration/removal and event. In particular, all listeners which have been registered in the initialization, or that the event thread has already invoked, *will* see the event.
- The listener event-handling methods can't throw checked exceptions. Any RuntimeException they throw will be wrapped by the manager with an instance of Error and re-thrown. The purpose of this policy is to reduce to minimum the risk of the application under test mistakenly catching listener exceptions. However, you should be aware that the possibility exists: the code under test may catch Errors, too, although this is not common (and not recommended!)
- Your listener code typically gets references to target program objects: objects read from or written to variables, array references, lock objects, and thread references. If you store these in data structures, beware of memory leaks - as long as they are stored by you the garbage collector can't reclaim the objects (which may in turn point to indefinitely large chunks of memory), unless you specially take care of it. One way to take care of it is to use weak references (see java.lang.ref.WeakReference and java.util.WeakHashMap).

- Beware of invoking methods of target program objects: these methods may be instrumented, and then you run the risk of ConTest/listener code called recursively (even infinitely). A special danger is to call toString(), hashCode
    - () or equals() on target program objects, which may override them and thus have this implementation instrumented. This is dangerous because you may do it indirectly and unknowingly: toString() is invoked if you output the object to an output stream or add it to a string. hashCode() and equals() are invoked if you use the object as a key in a hash map or set. Use reference equality testing, java.lang.System.identityHashCode(), java.util.IdentityHashMap, methods from com.ibm.contest.utils.Miscallenous, and (in view of the previous item) com.ibm.contest.runtime.utils.WeakIdentityHashMap.

        - You can safely call any method of Object (or any uninstrumented superclass of user's object) if it's final, such as getClass(), or Thread.getName(), since these cannot be overridden, and hence there is no danger of their being instrumented.
        - Program location argument is not a target program object, and you can safely print it, use it as key in a hash, etc. However, given the fact previously mentioned of the uniqueness of location instances, you may prefer, for efficiency, to use IdentityHashCode() and reference equality testing to hashCode() and equals(). The first two run in constant time, while the last two are linear in the string length (and program locations are rather long).
- Listeners code must not be given to ConTest to instrument - this may lead to infinite recursion. If the listener code is written as part of the target program, use instrumentation exclusion mechanisms.
- Listener code for general use should strive not to print to standard output or standard error: this can mix with target program output, and confuse possible automatic checking of the output (or even manual checking!). It is more robust to print output to a file. If printing fails - recall that listener code can't throw IOException outside - you have the option to either ignore or throw a RuntimeException which will result in an Error.
    - One kind of printing which *is* recommended is a verbose printing at program startup - the user can control whether such printing is done or not, and when it is allowed it is valuable in order to check configuration. See class VerbosePrinter.
- The implementation is extremely biased in favor of efficiency of invoking the listeners at runtime events. There is a big penalty in registering/removing listeners.
- If you want your listener to support ConTest seed replay (and normally, you should), you need to follow just two rules: (1) if there are user-controlled preferences which affect the behavior of the listener, they should be in ConTest properties file (see class ContestPropertiesFileReader. (2) If your listener makes choices based on a random value, this value should be taken from com.ibm.contest.utils.QuickRandom.

3.5. Infrastructure for Search Algorithms

The interface for search algorithm is implemented through a generic search algorithm for minimal noise. The algorithm is invoked through a call to the function `run(TestParams)`, which receives information about the specific program under test. The search algorithm provides the following capabilities:

- `getAllPoints()`: extracts the set of all possible locations of the program under test where we may want to add noise

- `runIteration(Set, int)`: instrument noise at a given subset of locations and run the instrumented program multiple times. Return the number of times a bug was found.

The search algorithm consists of two parts:

1. The initialization phase, which is implemented in the generic class and cannot be overridden. The initialization creates an output log and initializes the command that runs the Java program. This command is later used by `runIteration(Set, int)`.
2. The search phase, which is implemented by subclasses in `runSpecial()`. There are no rules regarding the implementation of `runSpecial`. A possible structure of such an implementation can be:
    a. Extract all instrumentation points, by calling `getAllPoints()`
    b. Test whether different subsets of instrumentation points induce the bug, by consecutive calls to `runIteration(Set, int)`
    c. Decide on a minimal set of points that induces the bug

4. Summary

This document describes the prototype tool for self-healing of concurrency-related failures for the development stage. The underlying algorithms are described in details in Deliverable 3.2.1.

ConTest infrastructures enabled us development of different failure analysis technologies and healing technology, and easy integration of these technologies into one prototype tool. Future improvements and enhancements to the tool will also benefit from this infrastructure. The Eclipse plug-in version of the tool enables easy development of such enhancements within the Eclipse Framework.